

Performance Markers & Blending, Anti Aliasing, Backface Culling

Adding Performance Markers

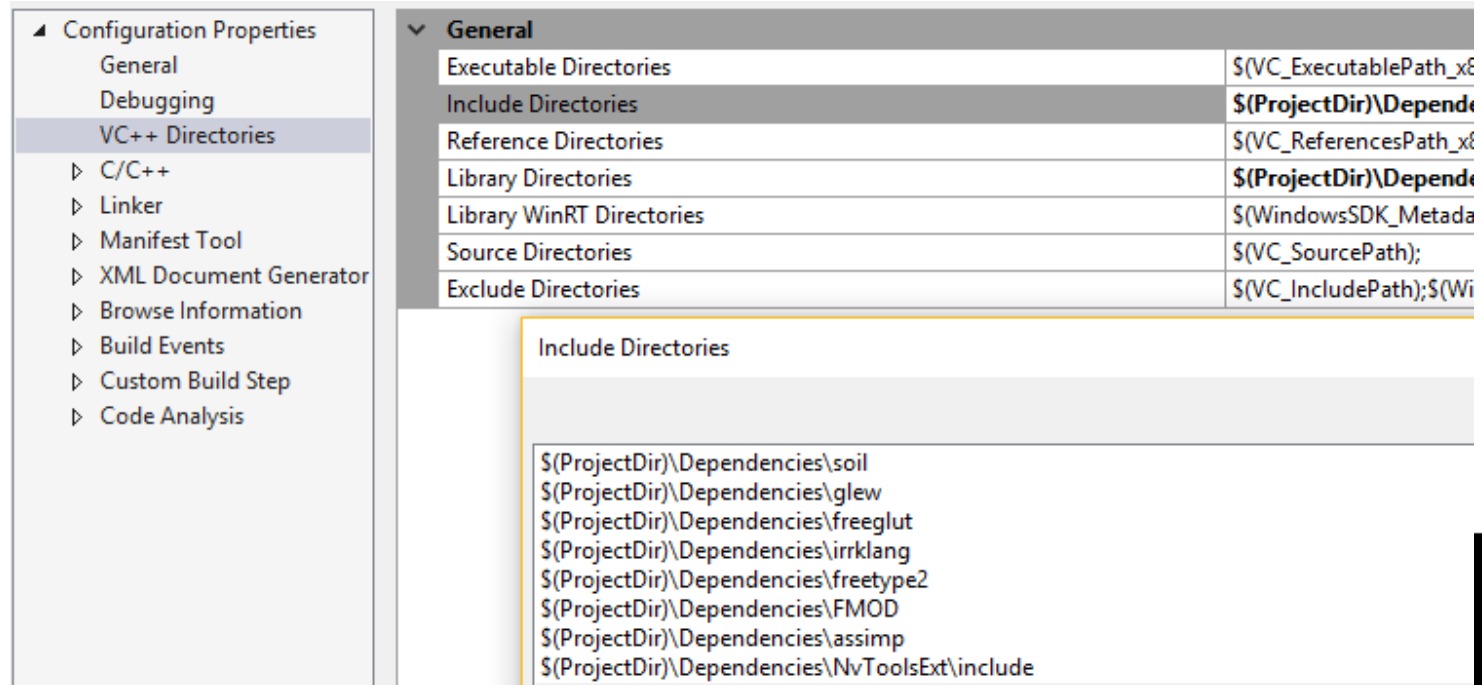
- Highlight a section of code while performance testing
- Add nSight Extensions library in Dependencies

gl-mds-tutorials > Advanced > Advanced_OpenGL > Advanced > Dependencies >

Name	Date modified	Type	Size
assimp	7/3/2017 8:50 PM	File folder	
FMOD	6/30/2017 12:00 PM	File folder	
freeglut	6/30/2017 12:00 PM	File folder	
freetype	6/30/2017 12:00 PM	File folder	
freetype2	6/30/2017 12:00 PM	File folder	
glew	6/30/2017 12:00 PM	File folder	
irrclang	6/30/2017 12:00 PM	File folder	
NvToolsExt	7/23/2017 12:27 PM	File folder	
soil	6/30/2017 12:00 PM	File folder	
ft2build.h	6/30/2017 12:00 PM	C/C++ Header	

Adding Performance Markers

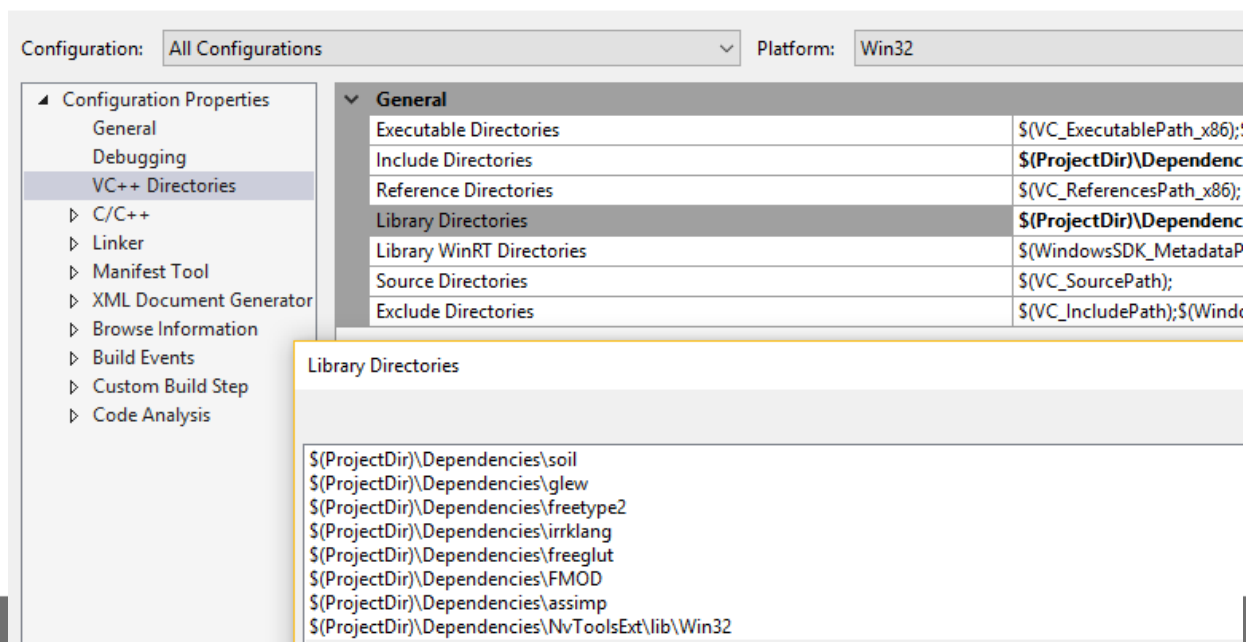
- Add extension include directory into VC++ Directories -> include Directory



Adding Performance Markers

- Add the library directory
- Specify the build version in the lib folder for library. Win32 or x64

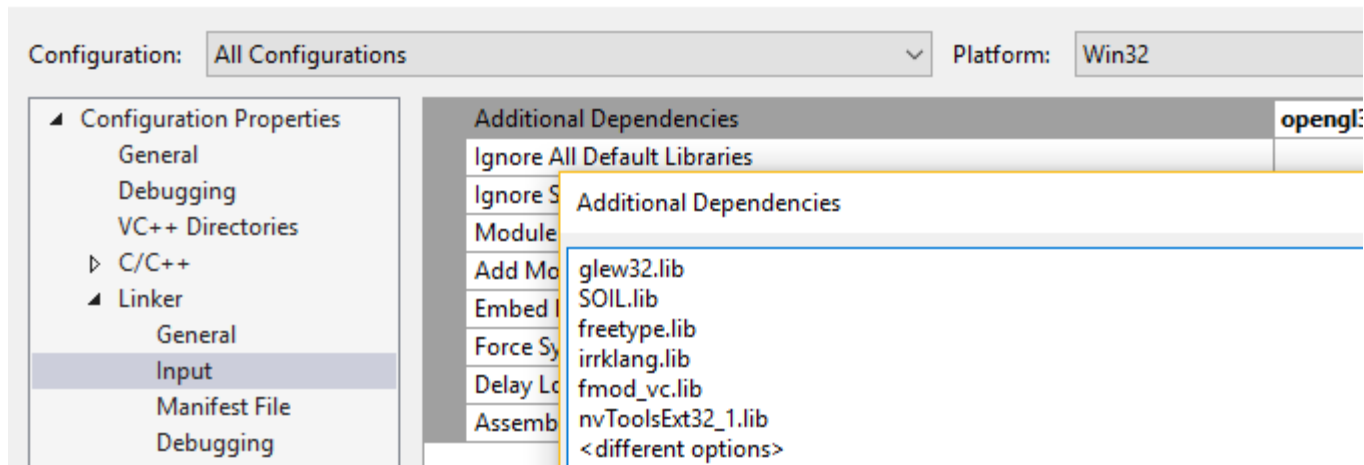
AdvancedOpenGL Property Pages



Adding Performance Markers

- In Linker->Input-> Additional Directories add nvToolsExt32_1.lib library.
- If developing for x64 add nvToolsExt64_1.lib instead.

AdvancedOpgl Property Pages

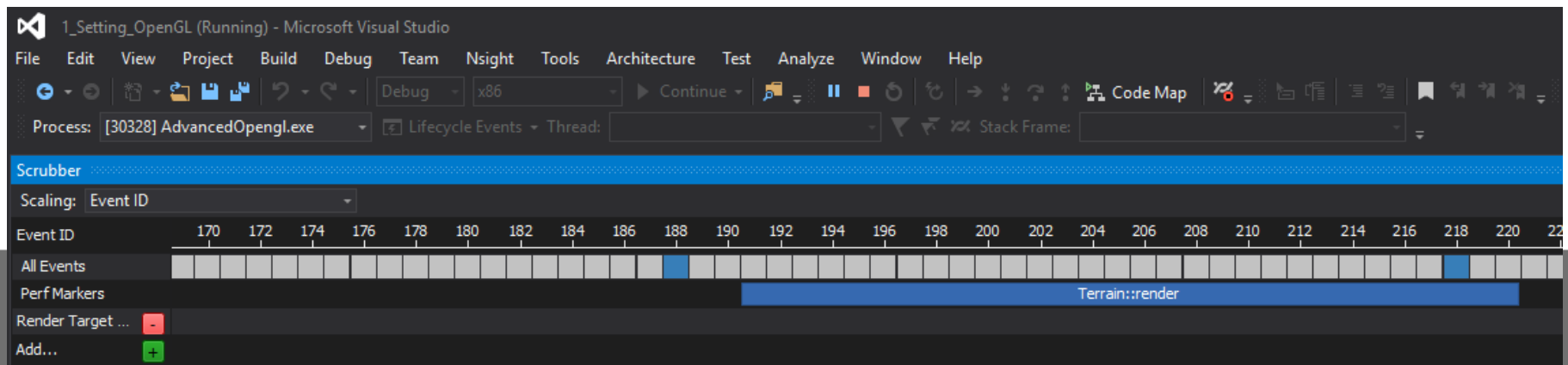


Usage

- For highlighting a certain piece of code, at the start add
nvtxRangePush(__FUNCTION__);
- The *__FUNCTION__* parameter will automatically look for the current function name
- And when you want to stop highlighting add *nvtxRangePop();*

Adding Performance Markers

- Run Graphic Debugging and pause
- In the Scrubber timeline you will the performance marker highlighting the start and stop of the marker
- Here the start and stop was specified before and after rendering the terrain.



Adding Custom Performance Markers

- If a more specific message needs to be displayed with a specific color a custom marker can be generated.
- Specifying an *eventAttribute* of type *nvtxEventAttributes_t*.
- Use *nvtxRangePushEx(&eventAttrib);* and pass in the attribute.
- When done call *nvtxRangePop();*

Usage

```
//push marker  
nvtxRangePushEx(&eventAttrib);  
  
shadowMap->renderFramebufferStart();  
tri->shadowMapPass();  
quad->shadowMapPass();  
cube->shadowMapPass();  
sphere->shadowMapPass();  
shadowMap->renderFramebufferEnd();  
  
//pop marker  
nvtxRangePop();
```

Adding Custom Performance Markers

- Specifying an attribute

//0 is the default for all attributes.

```
nvtxEventAttributes_t eventAttrib = { 0 };
```

// set the version and the size information

```
eventAttrib.version = NVTX_VERSION;
```

```
eventAttrib.size = NVTX_EVENT_ATTRIB_STRUCT_SIZE;
```

// configure the attributes.

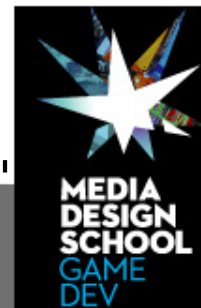
```
eventAttrib.colorType = NVTX_COLOR_ARGB;
```

```
eventAttrib.color = 0xFF00FF00; // green color
```

// custom message

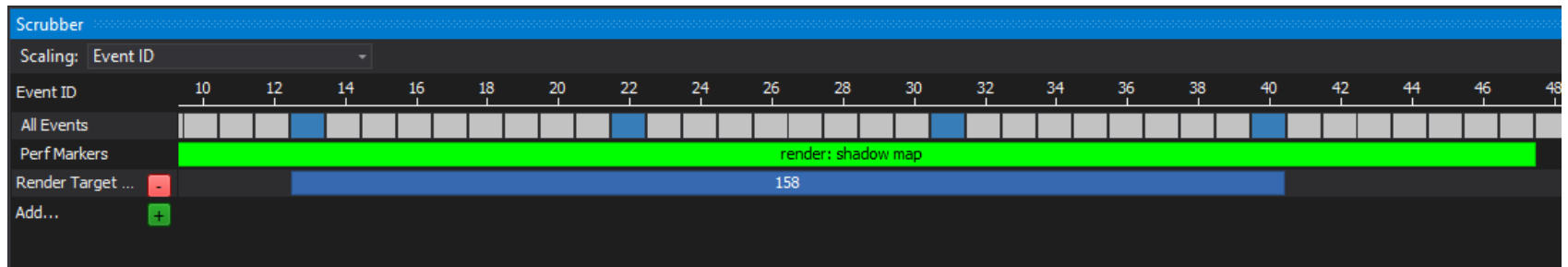
```
eventAttrib.messageType = NVTX_MESSAGE_TYPE_ASCII;
```

```
eventAttrib.message.ascii = __FUNCTION__ ": shadow map"
```



Adding Custom Performance Markers

- Output



- You can also have nested markers, specify categories, etc.
- Look at the documentation and add nested markers as exercise.

Blending

Blending

- Once an incoming fragment has passed all of the enabled fragment tests, it can be combined with the current contents of the color buffer in one of several ways.
- The default, is to overwrite the existing values
- Alternatively, you might want to combine the color present in the framebuffer with the incoming fragment color---a process called blending

Blending

- Blending is associated with the fragment's *alpha value*.
- Alpha is the fourth color component, and all colors in OpenGL have an alpha value
- it's a measure of translucency, and is what's used when you want to simulate translucent objects, like colored
- Glass or Water Surface for example

Blending

- Where color of a translucent object is a combination of that object's color with the colors of all the objects you see behind it
- OpenGL to do something useful with alpha, the pipeline needs more information than the current primitive's color (which is the color output from the fragment shader)
- It needs to know what color is already present for that pixel in the framebuffer

Blending

- In basic blending mode, the incoming fragment's color is linearly combined with the current pixel's color.
- *coefficients* control the contributions of each term
- those coefficients are called the *source-* and *destination-blending factors*

Blending

Final Colour = ($S_r R_s + D_r R_d$, //red

$S_g G_s + D_g G_d$, //green

$S_b B_s + D_b B_d$, //blue

$S_a A_s + D_a A_d$) //alpha

- (S_r, S_g, S_b, S_a) represent the source-blending factors. (D_r, D_g, D_b, D_a) represent the destination factors.
- (R_s, G_s, B_s, A_s) and (R_d, G_d, B_d, A_d) represent the colors of the source fragment and destination pixel respectively

Blending

- Controlling
 - Blending factors and
 - Blending Equation.

Blending

- Blending Factors
- You may call `glBlendFunc()` and choose two blending factors: the first factor for the source RGBA and the second for the destination RGBA
- `void glBlendFunc(GLenum srcfactor, GLenum destfactor);`

Blending

Constant	RGB Blend Factor	Alpha Blend Factor
GL_ZERO	$(0, 0, 0)$	0
GL_ONE	$(1, 1, 1)$	1
GL_SRC_COLOR	(R_s, G_s, B_s)	A_s
GL_ONE_MINUS_SRC_COLOR	$(1, 1, 1) - (R_s, G_s, B_s)$	$1 - A_s$
GL_DST_COLOR	(R_d, G_d, B_d)	A_d
GL_ONE_MINUS_DST_COLOR	$(1, 1, 1) - (R_d, G_d, B_d)$	$1 - A_d$
GL_SRC_ALPHA	(A_s, A_s, A_s)	A_s
GL_ONE_MINUS_SRC_ALPHA	$(1, 1, 1) - (A_s, A_s, A_s)$	$1 - A_s$
GL_DST_ALPHA	(A_d, A_d, A_d)	A_d

Blending

- Blending Equation
- With standard blending, colors in the framebuffer are combined (using addition) with incoming fragment colors to produce the new framebuffer color.
- `void glBlendEquation(GLenum mode);`

Blending

Blending Mode Parameter	Mathematical Operation
GL_FUNC_ADD	$C_sS + C_dD$
GL_FUNC_SUBTRACT	$C_sS - C_dD$
GL_FUNC_REVERSE_SUBTRACT	$C_dD - C_sS$
GL_MIN	$\min(C_sS, C_dD)$
GL_MAX	$\max(C_sS, C_dD)$

- GL_FUNC_ADD by default

Addition and Subtraction

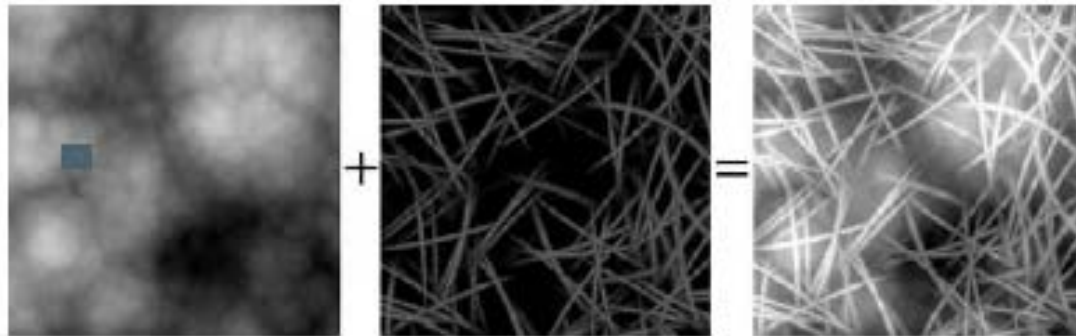


Figure 8.2: Adding source and destination color. Adding creates a brighter image since color is being added.

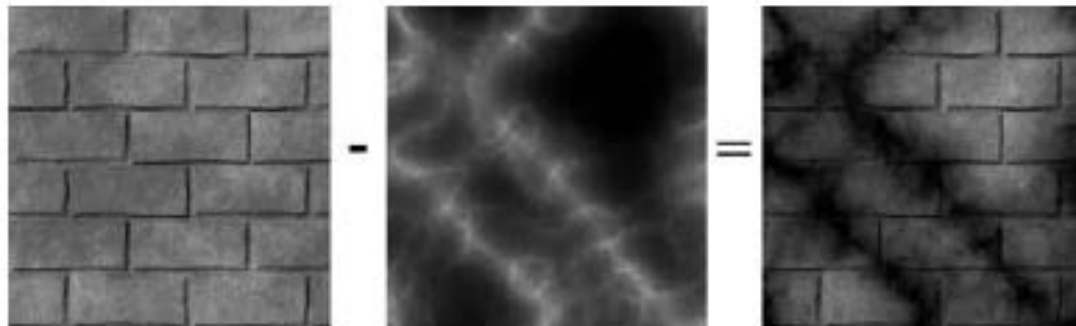


Figure 8.3: Subtracting source color from destination color. Subtraction creates a darker image since color is being removed.

Multiply

$$C = C_{src} \otimes F_{src} \boxplus C_{dst} \otimes F_{dst}$$

$$C = C_{src} \otimes (0, 0, 0) + C_{dst} \otimes C_{src}$$

$$C = C_{dst} \otimes C_{src}$$

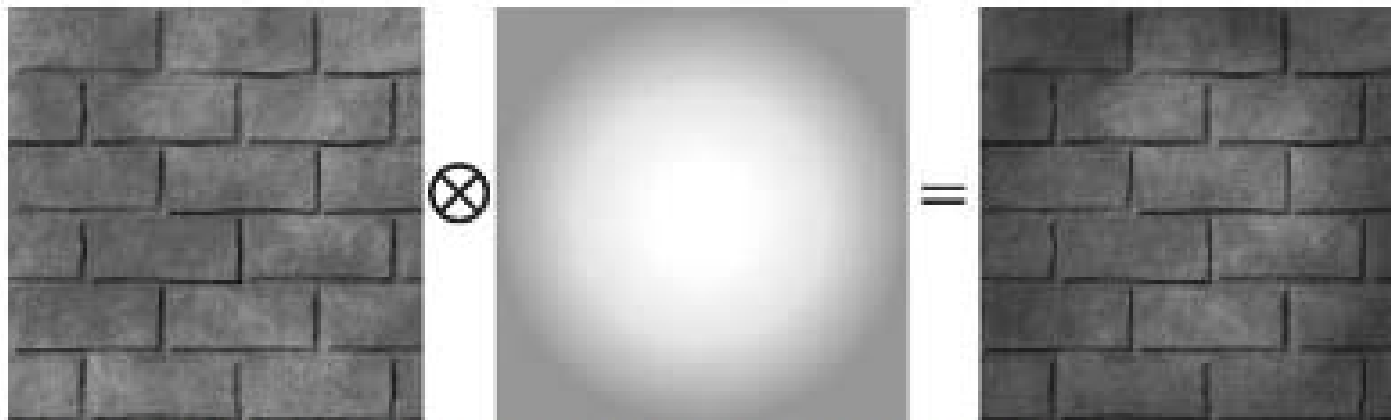


Figure 8.4: Multiplying source color and destination color.

Transparency

$$C = C_{src} \otimes F_{src} \boxplus C_{dst} \otimes F_{dst}$$

$$C = C_{src} \otimes (a_s, a_s, a_s) + C_{dst} \otimes (1 - a_s, 1 - a_s, 1 - a_s)$$

$$C = a_s C_{src} + (1 - a_s) C_{dst}$$

For example, suppose $a_s = 0.25$,

$$C = a_s C_{src} + (1 - a_s) C_{dst}$$

$$C = 0.25 C_{src} + 0.75 C_{dst}$$



Usage

- Render function

```
glEnable(GL_BLEND);
```

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

```
glBindVertexArray(vao);
```

```
glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, 0);
```

```
glBindVertexArray(0);
```

```
glDisable(GL_BLEND);
```

Notes

- While loading the texture using SOIL load the alpha channel data as well.
- `unsigned char* image = SOIL_load_image(texFileName.c_str(), &width, &height, 0, SOIL_LOAD_RGBA);`
- `glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE, image);`

Notes

- The object with the transparency needs to be rendered at the end after rendering all other objects.
- JPEG doesn't support transparency. You'll need to stick to PNG or GIF.
- So the water texture was stored as a PNG for the effect to take place.

Anti Aliasing

Anti Aliasing

- Because the pixels on a monitor are not infinitely small, an arbitrary line cannot be represented perfectly on the computer
- stair-step" (aliasing) effect, which can occur when approximating a line by a matrix of pixels monitor



Anti Aliasing

- Shrinking the pixel sizes by increasing the monitor resolution can alleviate the problem significantly.
- When increasing the monitor resolution is not possible or not enough, we can apply antialiasing techniques.
- Supersampling
- Multisampling

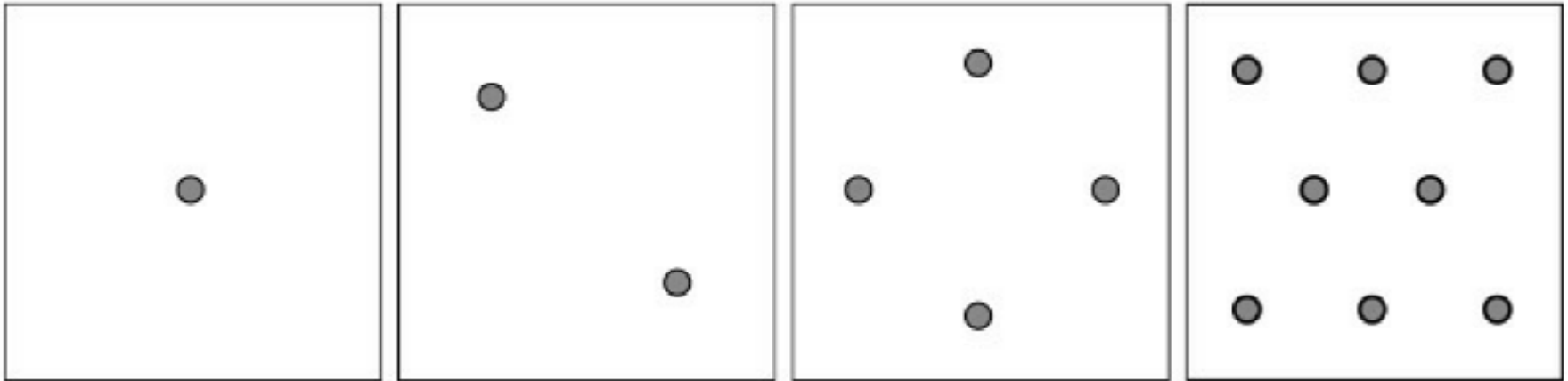
Anti Aliasing

- *supersampling*, works by making the back buffer and depth buffer 4X bigger than the screen resolution.
- The 3D scene is then rendered to the back buffer at this larger resolution
- when it comes time to present the back buffer to the screen, the back buffer is resolved (or downsampled) such that 4 pixel block colors are averaged together to get an averaged pixel color
- Supersampling is **expensive** because it increases the amount of pixel processing and memory by fourfold

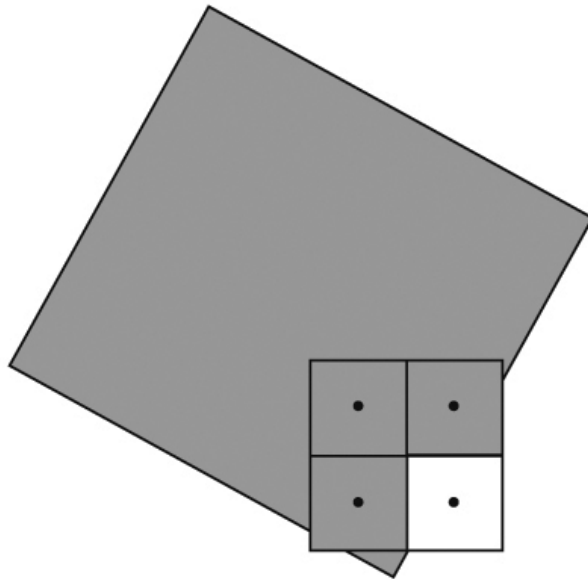
Anti Aliasing

- To increase the sample rate of the image, store multiple samples for every pixel on the screen.
- This technique is known as multi-sample antialiasing (MSAA).
- Rather than sampling each primitive only once, OpenGL will sample the primitive at multiple locations within the pixel and, if any are hit, run your shader.

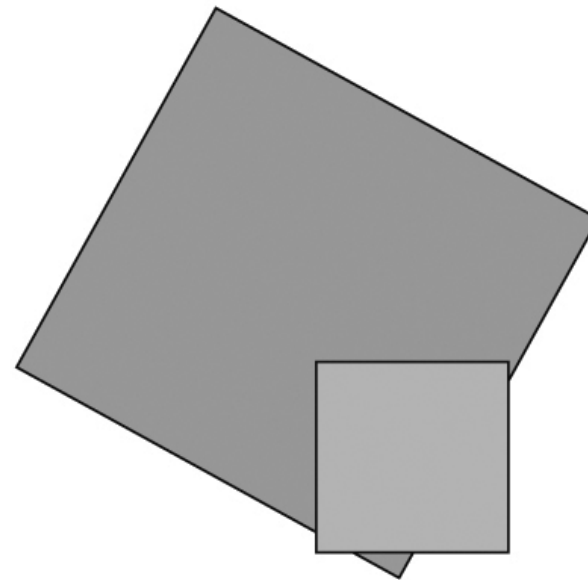
Anti Aliasing



Anti Aliasing



(a)



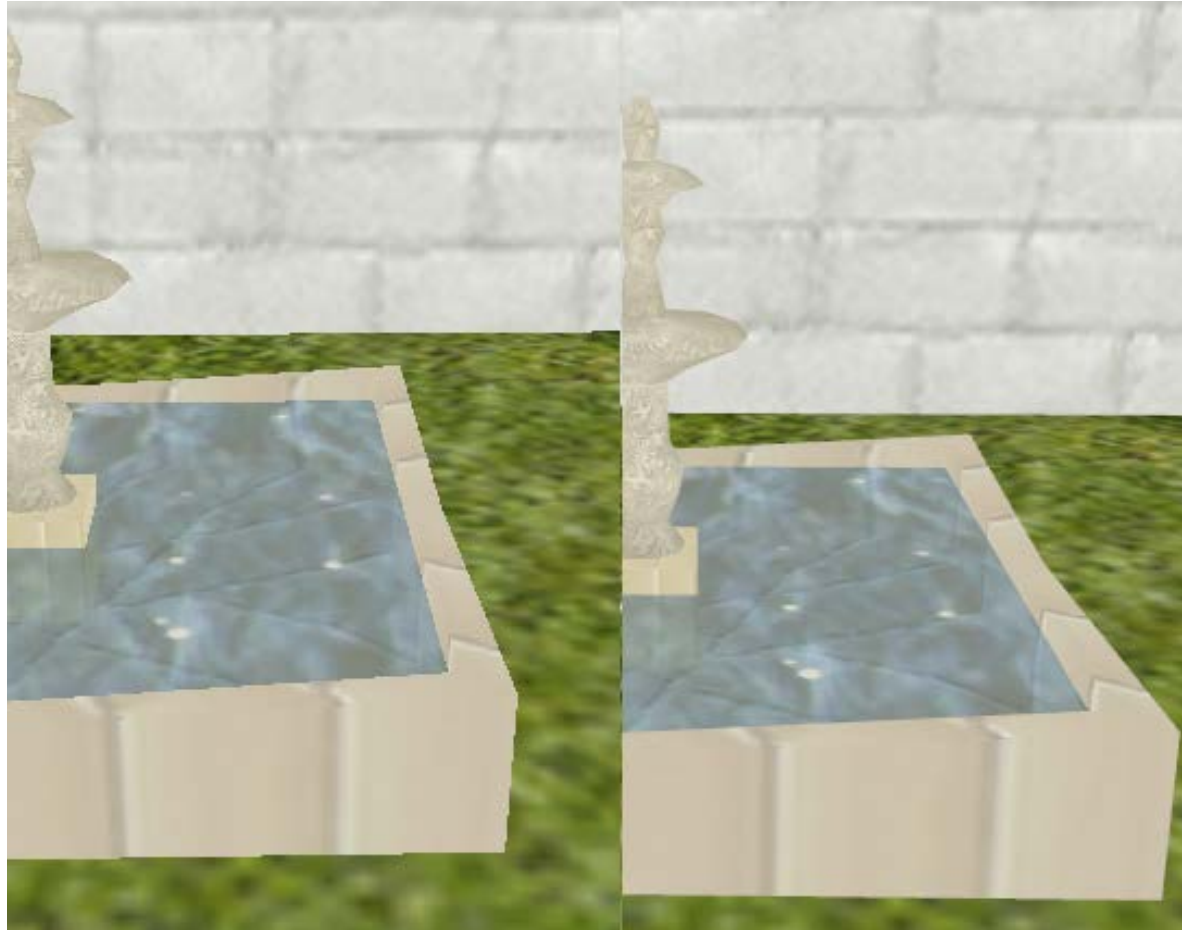
(b)

Usage

- In the init function at the start of the game.
- Initialize GLUT parameters in Main
- `glutInitDisplayMode(GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA | GLUT_MULTISAMPLE);`
- Set multisample level

```
glutSetOption(GLUT_MULTISAMPLE, 8);  
glEnable(GL_MULTISAMPLE);
```

Output



Backface Culling

Backface Culling

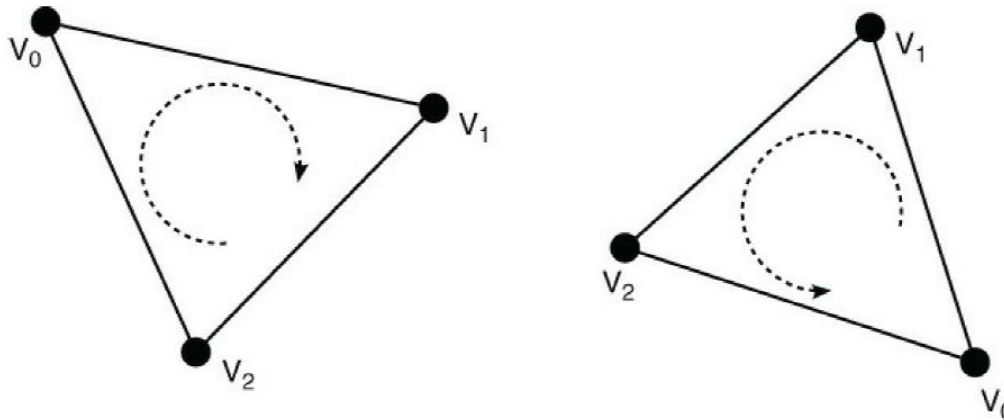
- The elimination of graphics primitives that would not be seen if rendered.
- Back-face culling eliminates the front or back face of a primitive so that the face isn't drawn.
- Frustum culling eliminates whole objects that would fall outside the viewing frustum.

Backface Culling

- If the triangle faces toward the viewer, then it is considered to be *frontfacing*;
- otherwise, it is said to be *back-facing*
- It is very common to discard triangles that are back-facing because when an object is closed, any back-facing triangle will be hidden by another front-facing triangle.

Backface Culling

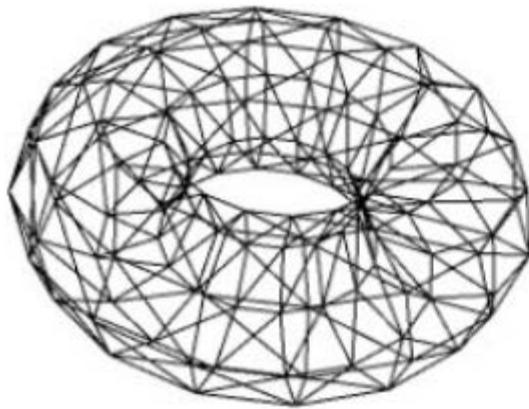
- Winding order determines if the face is front facing or back



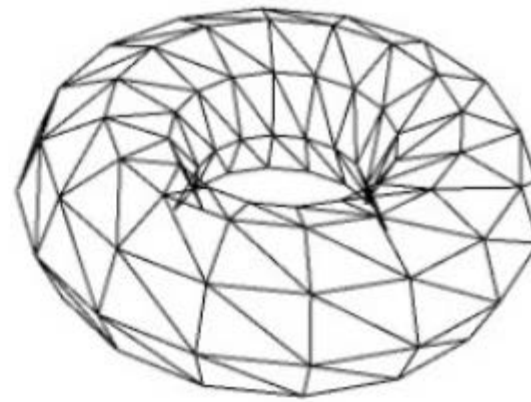
- `glFrontFace(GL_CCW);`
- Counterclock Wise by DEFAULT

Backface Culling

- Set the back faces to be culled.
- `glCullFace(GL_BACK);`



Torus drawn in wire-frame
without back face culling



Torus drawn in wire-frame
with back face culling

Backface Culling

- Enable face culling
- `glEnable(GL_CULL_FACE);`

- Example

```
glEnable(GL_CULL_FACE);
```

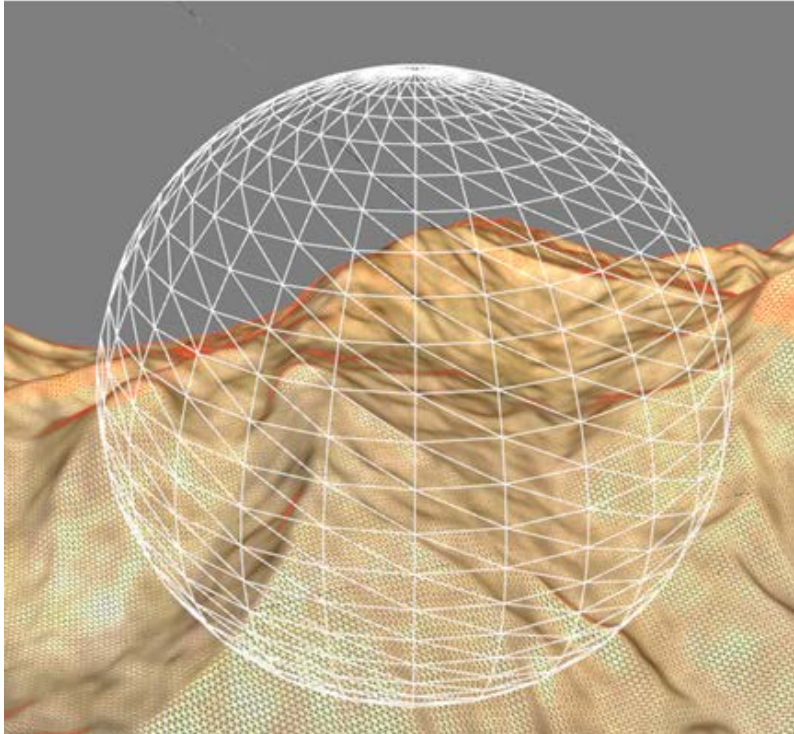
```
glBindVertexArray(vao);
```

```
glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, 0);
```

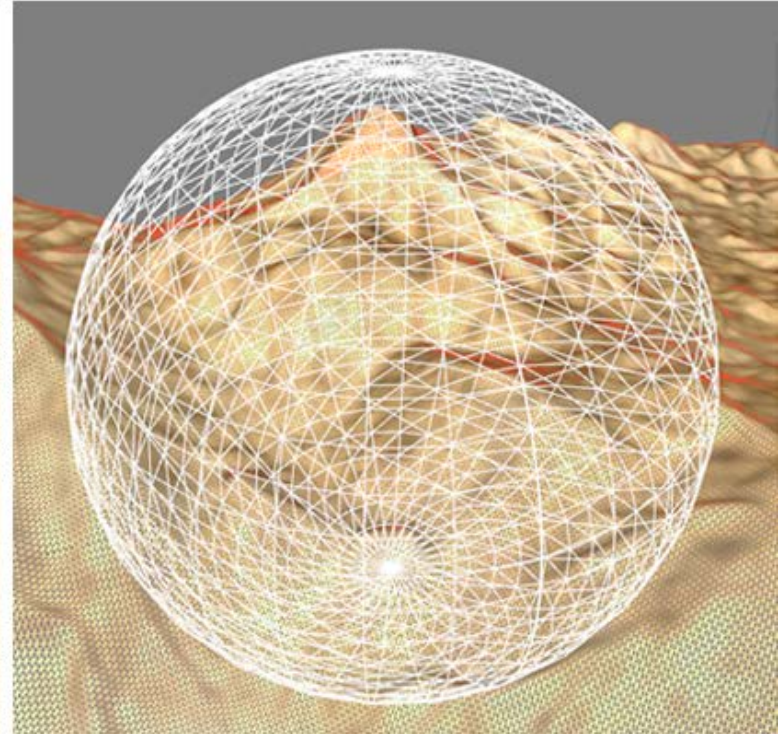
```
glBindVertexArray(0);
```

```
glDisable(GL_CULL_FACE);
```

Output



Enabled



Disabled

Exercises

- Add blending
- Animate the textures water surface
- Enable multisampling
- Enable back face culling for objects